Integers

Number Representation

Decimal, Binary, Hexadecimal

$$1209_{[10]} = 1\times10^3 + 2\times10^2 + 0\times10^1 + 9\times10^0$$

$$100101_{[2]} = 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$

$$B0A_{[16]} = B\times16^2 + 0\times16^1 + A\times16^0$$

base

position of digit

Hexadecimal

- $0_{[16]}$ $0000_{[2]}$ $0_{[10]}$
- $1_{[16]}$ $0001_{[2]}$ $1_{[10]}$
- $2_{[16]}$ $0010_{[2]}$ $2_{[10]}$
- $3_{[16]}$ $0011_{[2]}$ $3_{[10]}$
- $4_{[16]}$ $0100_{[2]}$ $4_{[10]}$
- $5_{[16]}$ $0101_{[2]}$ $5_{[10]}$
- $6_{[16]}$ $0110_{[2]}$ $6_{[10]}$
- $7_{[16]}$ $0111_{[2]}$ $7_{[10]}$
- $8_{[16]}$ $1000_{[2]}$ $8_{[10]}$
- $9_{[16]}$ $1001_{[2]}$ $9_{[10]}$
- $A_{[16]}$ $1010_{[2]}$ $10_{[10]}$
- $B_{[16]}$ $1011_{[2]}$ $11_{[10]}$
- $C_{[16]}$ $1100_{[2]}$ $12_{[10]}$
- $D_{[16]}$ $1101_{[2]}$ $13_{[10]}$
- $E_{[16]}$ $1110_{[2]}$ $14_{[10]}$
- $F_{[16]}$ $1111_{[2]}$ $15_{[10]}$

Fixed-size Number Representation

Binary Arithmetic

```
      1
   1010   (10)
+  1010   (10)
---------
  10100   (20)
```

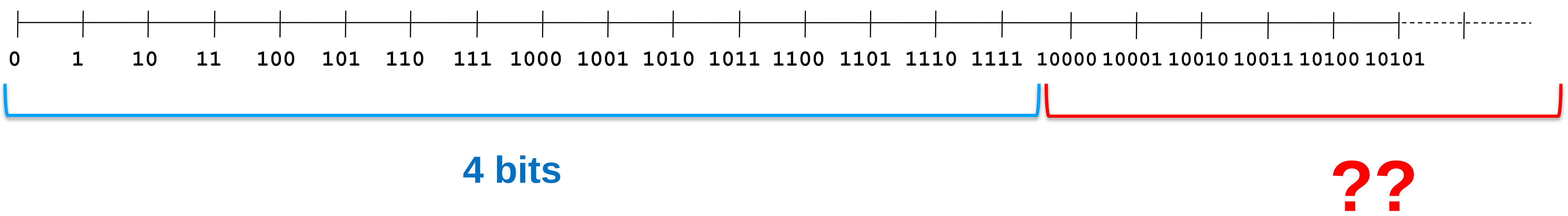
```
     110    (6)
  × 1010   (10)
----------
       0
     110
      0
+  110
----------
  111100   (60)
```

- *What if we have only 4 binary digits to represent integers?*

Binary Arithmetic … on 4-bit Words

```
        1
     1010   (10)
+    1010   (10)
---------
    10100   (20)
```

```
      110    (6)
×    1010   (10)
---------
        0
     110
    0
+ 110
---------
   111100   (60)
```



Fixed-size Representation

32 bits in C0

- Allows efficient operations in hardware
- We have to handle **overflow**
  - Raise error/exception
  - Something else …

Handling Overflow as Error

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_B
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(T
end if;
P_M_DERIVE(T_ALG.E_BH) :=
    UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M
```

Ariane 5

Handling Overflow as Error

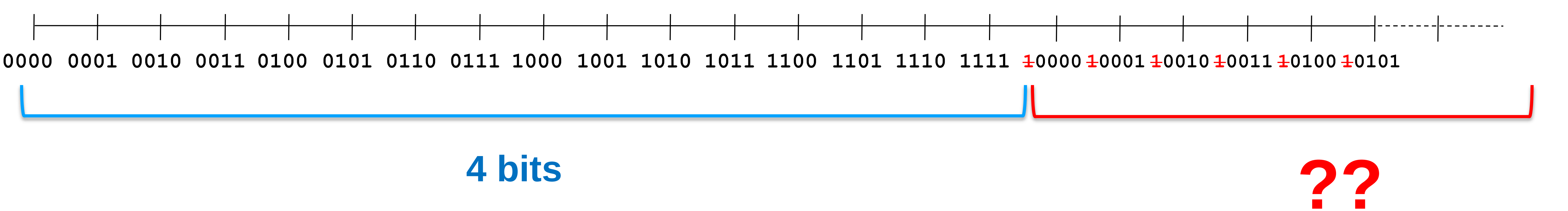
- Hard to reason about code
  - $n + (n - n)$ and $(n + n) - n$ are equal in math …
- … but with fixed size numbers,
  - $n + (n - n)$ always equal to n
  - $(n + n) - n$ may overflow

We want to be able to use the laws of arithmetic

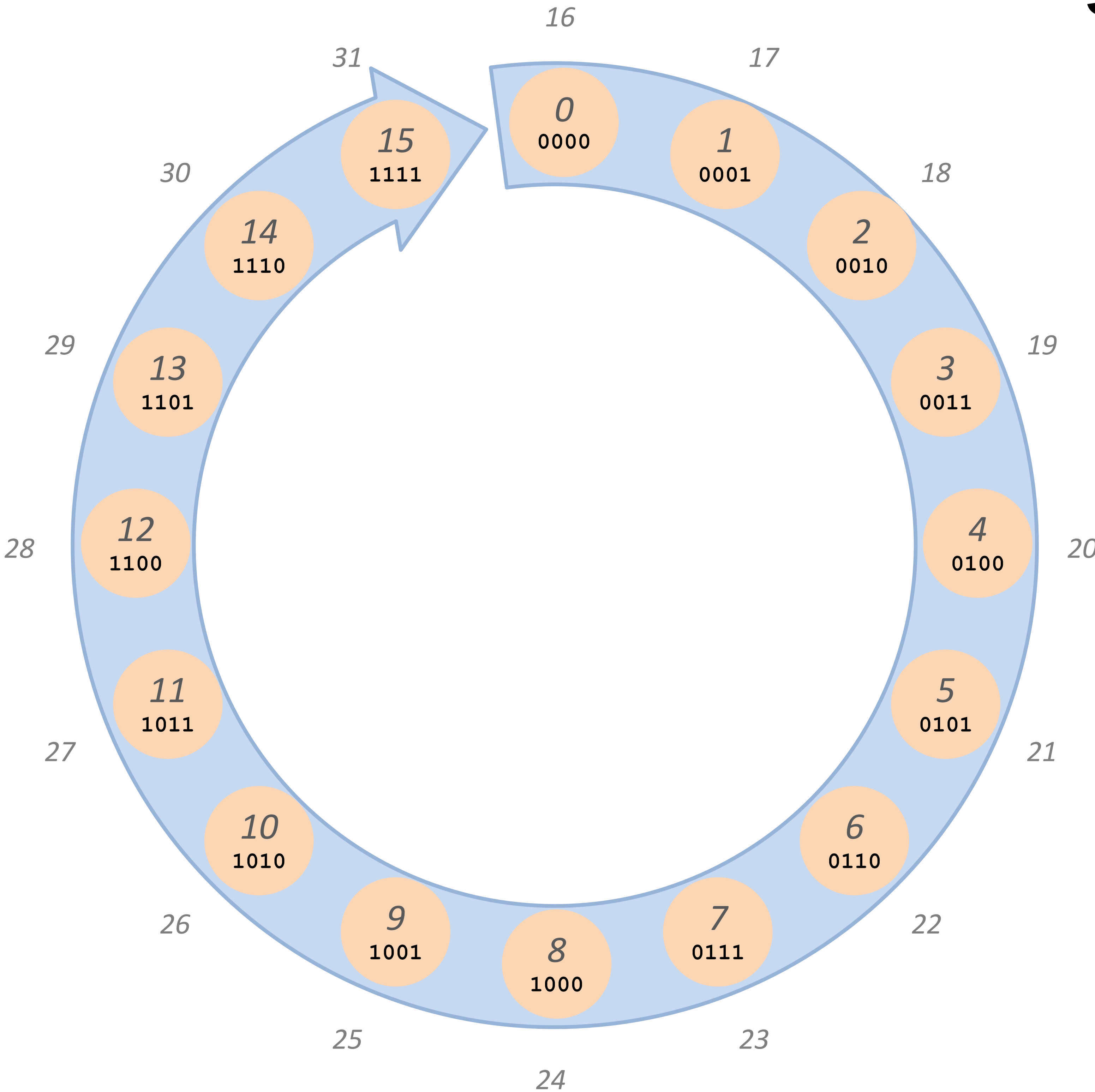
Modular Arithmetic

```
       1
    1010   (10)
+   1010   (10)
--------
   ~~1~~0100  (20)
```

```
     110    (6)
 ×  1010   (10)
---------
       0
     110
     0
+  110
---------
~~11~~1100    (60)
```



Integers Modulo 16



- From number line to a number circle
- Addition is moving clockwise
- Arithmetic mod 16 (= 2^4), corresponds to a fictional machine with word size 4

Laws of Modular Arithmetic

$x + y = y + x$	Commutativity of addition
$(x + y) + z = x + (y + z)$	Associativity of addition
$x + 0 = x$	Additive unit
$x * y = y * x$	Commutativity of multiplication
$(x * y) * z = x * (y * z)$	Associativity of multiplication
$x * 1 = x$	Multiplicative unit
$x * (y + z) = x * y + x * z$	Distributivity
$x * 0 = 0$	Annihilation

Same laws as traditional arithmetic!

Reasoning about int`s

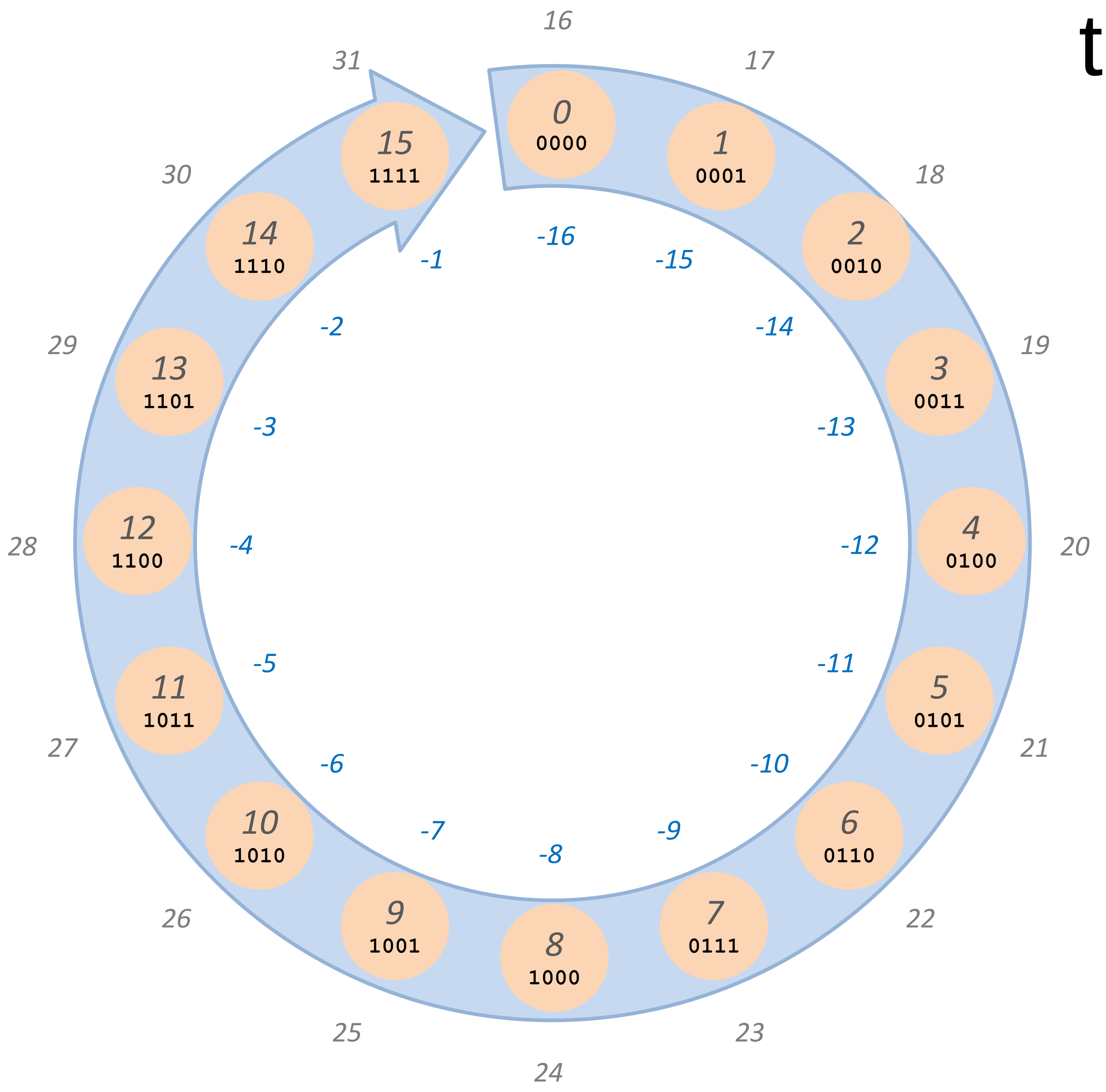
```
string foo(int x) {
  int z = 1+x;
  if (x+1 == z)
    return "Good";
  else
    return "Bad";
}
```

This is equivalent to
x+1 == 1+x
by substitution

x+1 == 1+x
is always **true**
by commutativity of addition

… so foo always returns "Good"

What about the Negatives?



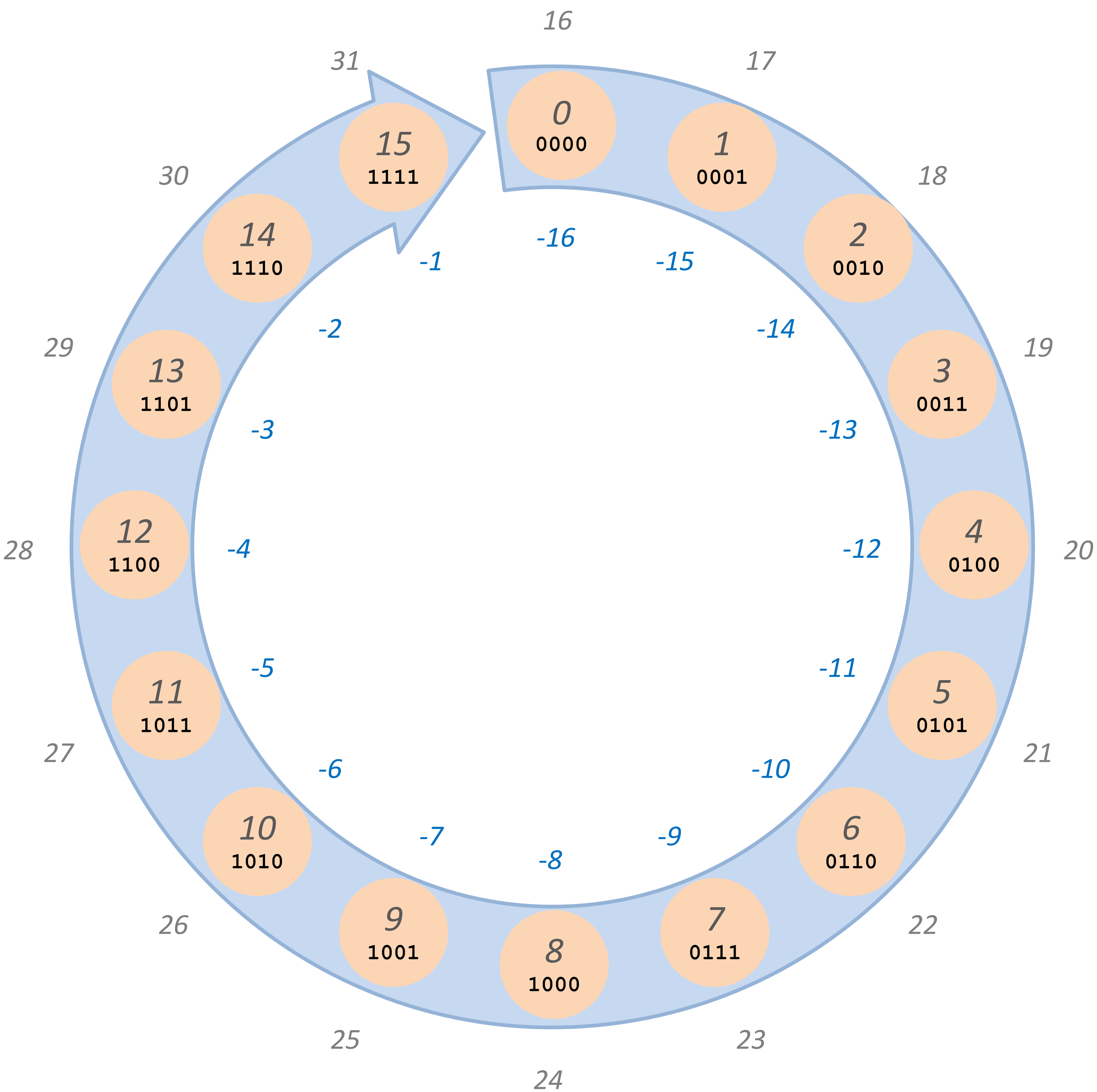
Subtraction

- $x - y$ is stepping y times counter-clockwise from x
- Define $-x = 0 - x$
- Then,

$x + (-x) = 0$	Additive inverse
$-(-x) = x$	Cancelation

Same laws as traditional arithmetic!

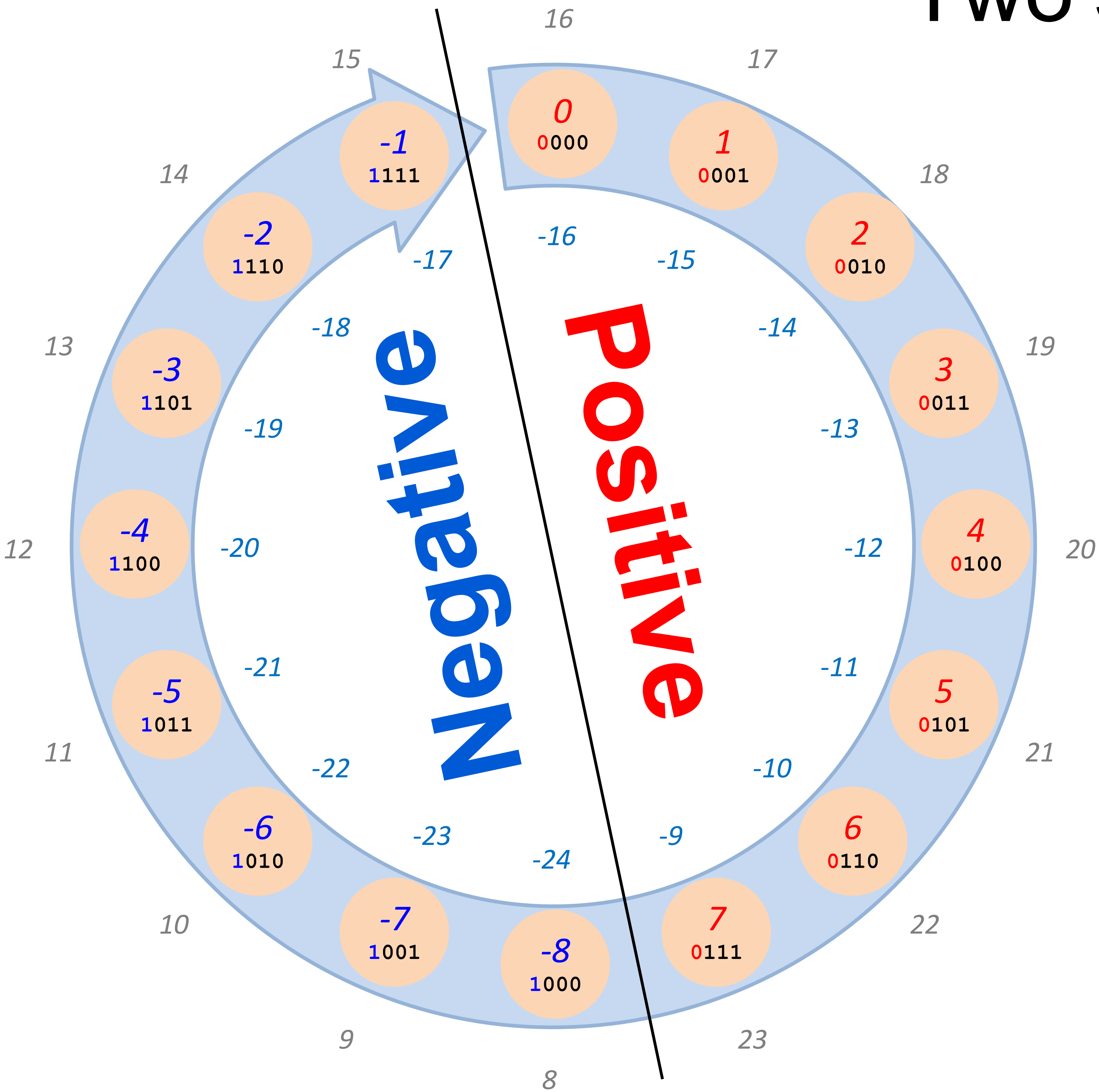
Two’s Complement



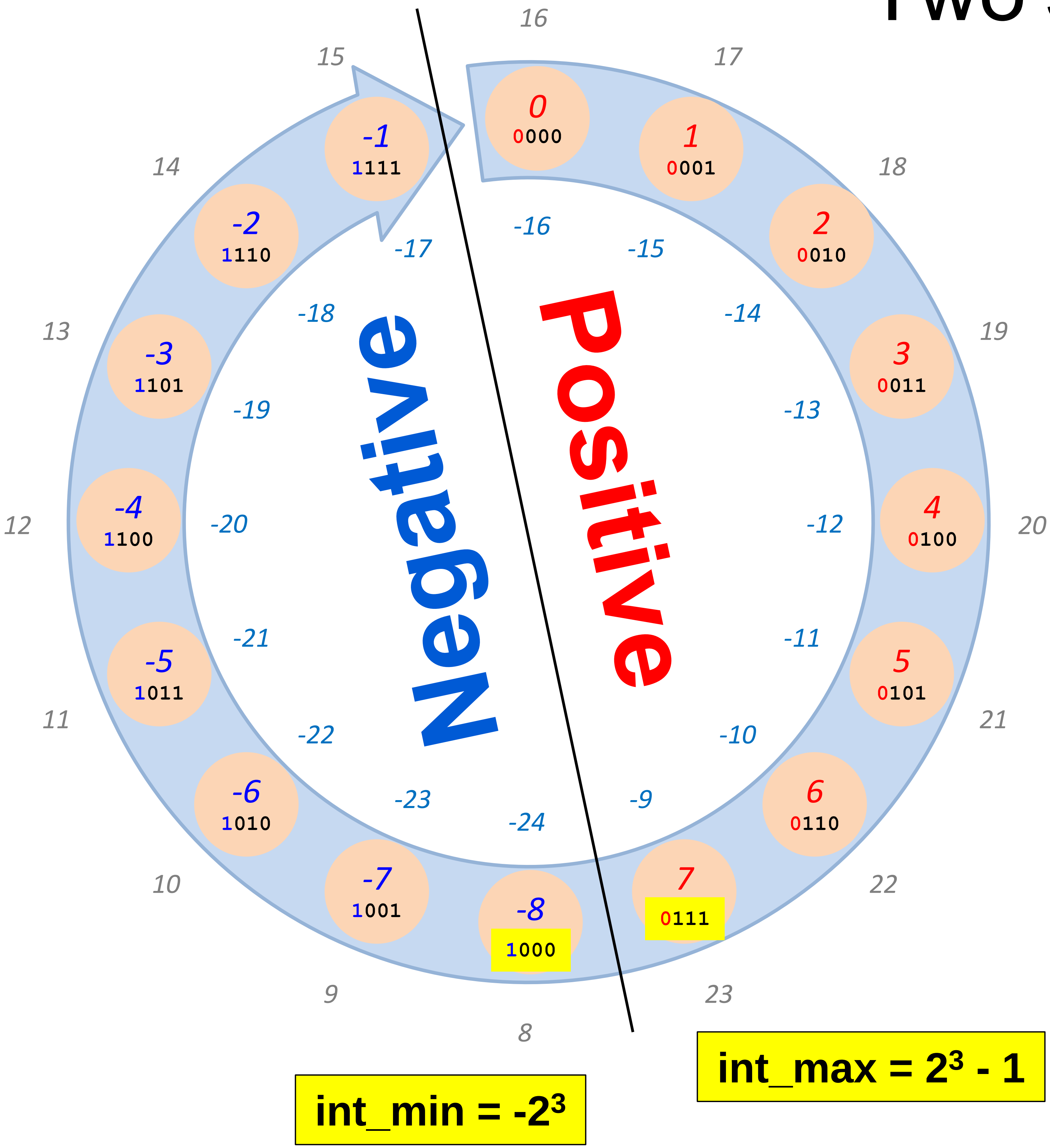
Rendering

- How should the computer print back to us **0100**?
  - 4? 20? -12?
- What about **1101**?
  - 13? 29? -3?

Two's Complement



Two's Complement



- With k bits
  - int_max = 2k - 1
  - int_min = -2k
- Off by one because of 0
- We can now talk about **ordering**
  - x < y, x ≤ y, …

Reasoning about int`s

```
string bar(int x) {
  if (x+1 > x)
    return "Good";
  else
    return "Strange";
}
```

When is x+1 **not** larger than x in C0?

Division and Modulus

- In calculus, *(x/y)* is *z* such that *y * z = x*
- Introduce a new operation to pick up the slack: **modulus**
  - *(x/y) * y + (x%y) = x*
  - *0 <= |x % y| < |y|*
- *x/y* rounds down for positive *x* and *y*
- What should *(x/y)* round down to for negative numbers?
  - C0 rounds “down” to 0
  - Python rounds towards -∞

But there is **no** int z such that 2 * z = 3

This doesn’t work for the integers

Safety Requirements

- Division by 0 is undefined (same for modulus)
  - Any time we have x/y in a program, we must have a reason to believe that y != 0
  - This is a safety requirement
  - x/y and x%y have *preconditions*

```
//@requires y != 0;
//@requires !(x == int_min() && y == -1);
```

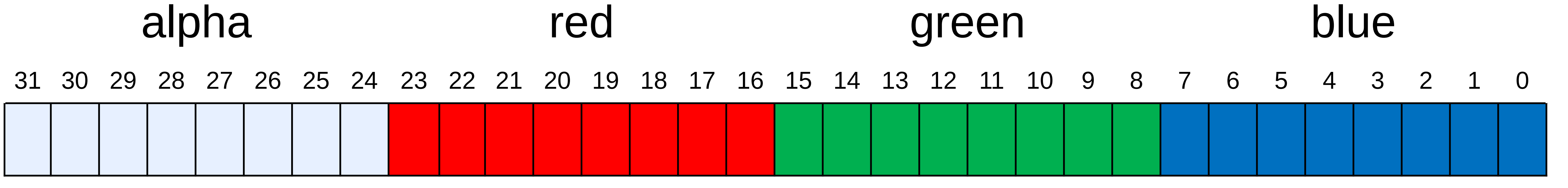
  - because chips raise errors on these inputs

Bit Patterns

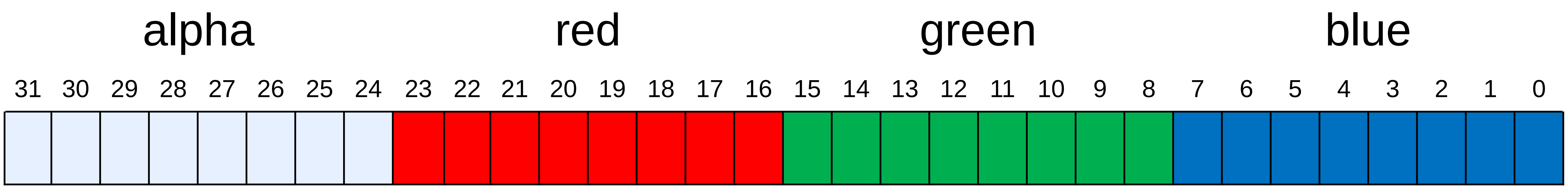
Bit Patterns

- use int to represent data other than numbers
  - pixels
  - network packets
  - …
- New set of operations to manipulate them
  - bitwise operators
  - shifts

Pixels as 32-bit int's (ARGB)

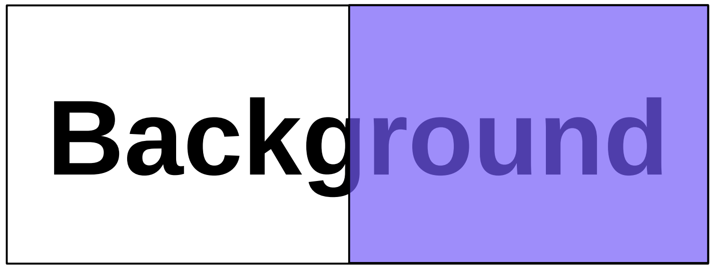


Example: Pixel



`1011 0011 0111 0011 0101 1010 1111 1001`

`B 3 7 3 5 A F 9`



Bitwise Operations

and

&	0	1
0	0	0
1	0	1

or

|	0	1
0	0	1
1	1	1

xor

^	0	1
0	0	1
1	1	0

not

~	0	1
	1	0

Bitwise Operations

- Apply to `int`'s, position by position
  - examples with just 4 bits

```
    1010          1010
  & 1001        | 1001
  ------        ------
    1000          1011
```

- Related to `&&` and `||` but not interchangeable
  - take int's as input, not bool's

Bitwise Operations

b

&	0	1
0	0	0
1	0	1

mask

0, always

same as b

b

|	0	1
0	0	1
1	1	1

same as b

1, always

b

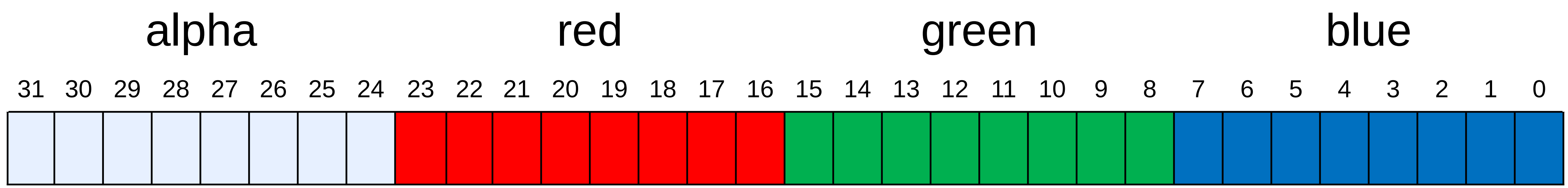
^	0	1
0	0	1
1	1	0

same as b

inverse of b

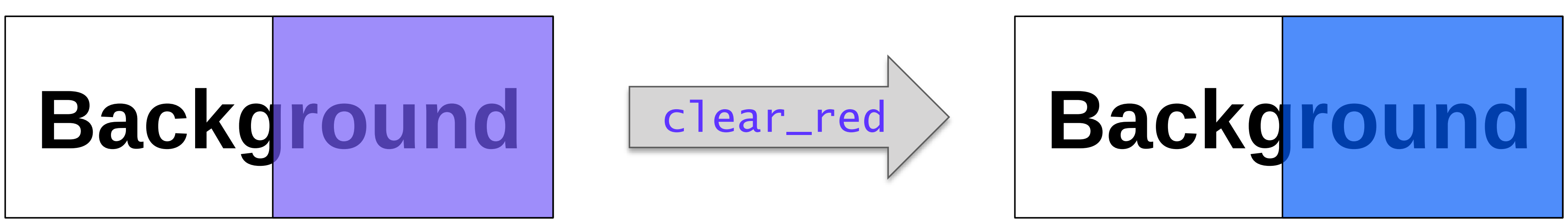
~	0	1
	1	0

Clearing Bits

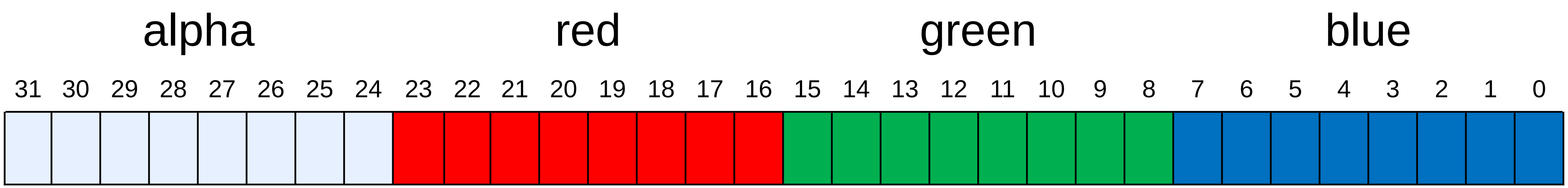



```
int clear_red(int p)
 {
  return p &
 0xFF00FFFF;
}
```

Mask

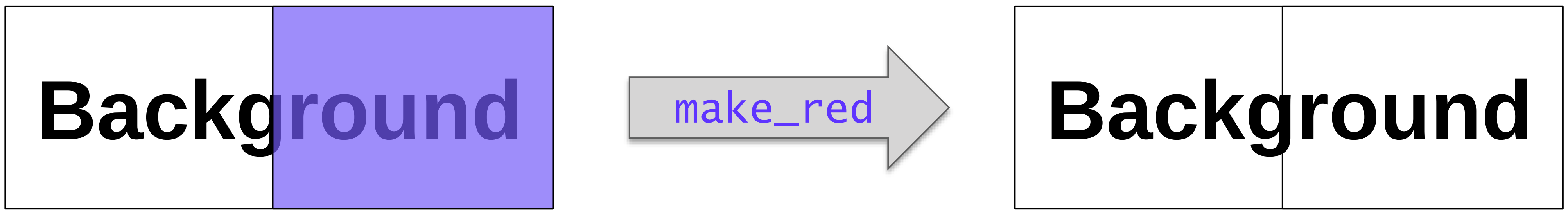


Isolating Red

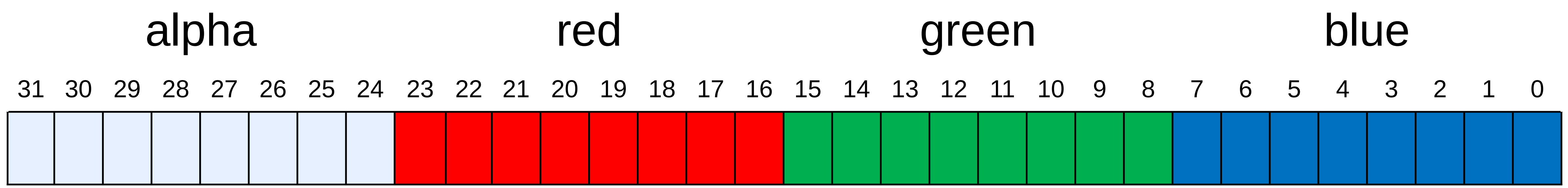



```
int make_red(int p) {
   int red = p & 0x00FF0000;
  return red;
}
```

Mask

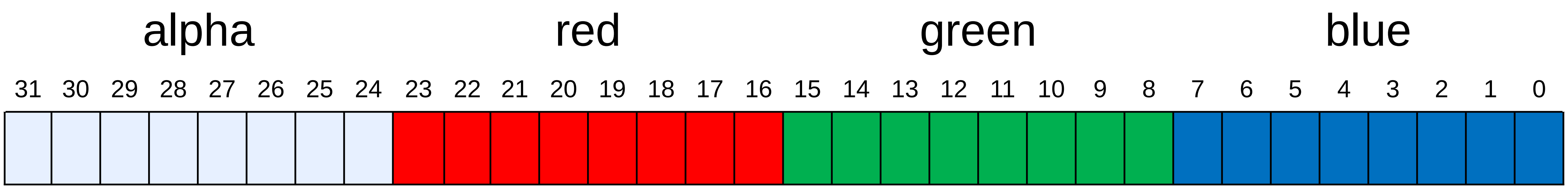


Example: Opacify




```
int opacify(int p) {
return p | 0xFF000000;
}
```

What does this Function do?




```
int franken_pixel(int p, int q) {
   int p_green = p & 0x0000FF00;
   int q_others = q & 0xFFFF00FF;
  return p_green | q_others;
}
```

- shifts `x` by `k` bits to the right
  - k rightmost bits are dropped
  - k leftmost bits are a ***copy*** of the leftmost bit
    - **sign extension**
- 0101 >> 1 = 0010
- 0101 >> 3 = 0000
- 1010 >> 1 = 1101
- 1010 >> 3 = 1111

Shifts: Moving Bits Around

Left shift: `x << k`

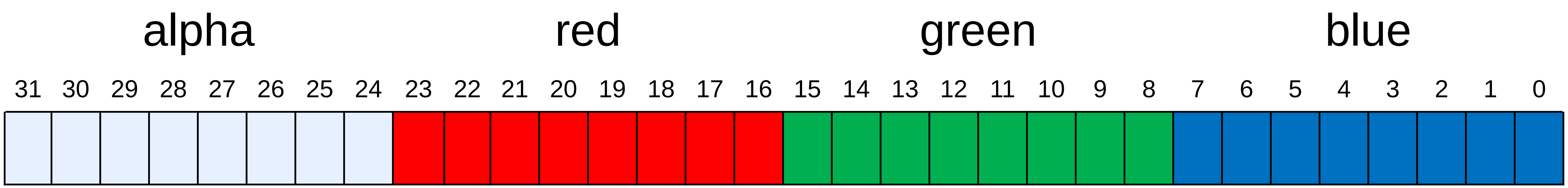
- shifts `x` by `k` bits to the left
  - k leftmost bits are dropped
  - k rightmost bits are 0
- 0101 << 1 = 1010
- 0101 << 3 = 1000

Right shift: `x >> k`

- shifts `x` by `k` bits to the right
  - k rightmost bits are dropped
  - k leftmost bits are a ***copy*** of the leftmost bit
    - **sign extension**
- 0101 >> 1 = 0010
- 0101 >> 3 = 0000
- 1010 >> 1 = 1101
- 1010 >> 3 = 1111

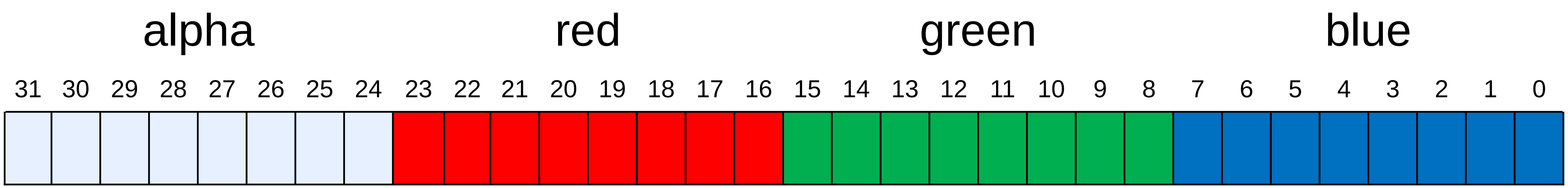
Preconditions: `//@requires 0 <= k && k < 32;`

Red Everywhere




```
int red_everywhere(int p) {
   int alpha = p & 0xFF000000;
   int red = p & 0x00FF0000;
  return alpha | red | (red >> 8) | (red >> 16);
}
```

Swapping the Alpha and Red Channels

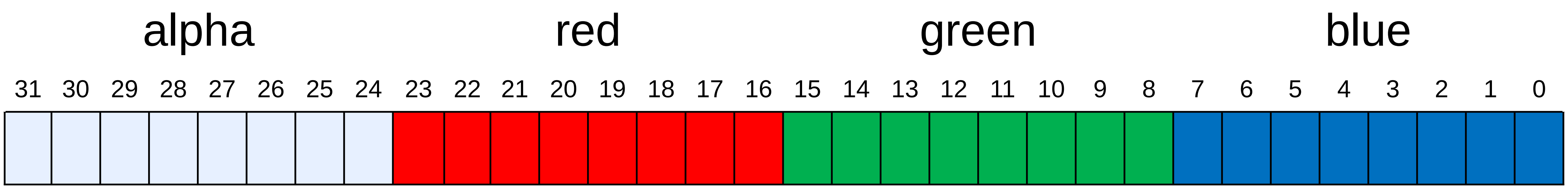



```
int BAD_swap_alpha_red(int p) {
   int new_alpha   = (p & 0x00FF0000) << 8;
   int new_red     = (p & 0xFF000000) >> 8;
   int old_green   = p & 0x0000FF00;
   int old_blue    = p & 0x000000FF;
 return new_alpha | new_red | old_green | old_blue;
}
```

What if the first bit is **1**?

Why is this function bad?

Swapping the Alpha and Red Channels




```
int swap_alpha_red(int p) {
   int new_alpha    = (p << 8) & 0xFF000000;
   int new_red      = (p >> 8) & 0x00FF0000; // fixed
   int old_green    = p & 0x0000FF00;
   int old_blue     = p & 0x000000FF;
 return new_alpha | new_red | old_green | old_blue;
}
```